MOTION IMAGERY STANDARDS BOARD	MISB TRM 1006
Technical Reference Material	**13 July 2012**
Key-Length-Value (KLV) Users Guide	

1 Why KLV

The Problem

All communications channels have bandwidth limitations; thus, efficient methods for representing data traveling over a bandwidth-constrained channel that are robust to both loss and corruption are necessary. Representation of data must also allow for the "uneducated" application. Ideally, even an application with limited knowledge regarding the data should be provided with sufficient information to do so anyway.

Further complicating matters is the remote platform. Different platforms have different ways of representing primitive data types. A big-endian integer, for example, will have a very different value on a little-endian machine. Similar issues exist for floating point, strings, and many other primitive data types. The data-in-motion representation must allow for and provide a way to deal with these issues.

Solutions

Techniques developed to represent data in motion are called encodings. There are three common approaches to representing (encoding) data for transport: unstructured, structured and semi-structured.

Free text is a common example of unstructured data. It has the advantage of being very flexible and very robust. Free text can express almost any concept in a manner that is comprehensible to most people. Even if sections of the message are corrupt or missing, the concept can still get through. However, unstructured data is very difficult for computers to process.

Structured data have a well-defined and predictable structure and meaning. National Imagery Transmission Format (NITF) is an example of structured data. Structured data are an effective way to exchange complex information between computer systems. Missing or corrupted data, however, can throw off the parsing algorithm, making the data unusable. While efficient, structured approaches are also brittle.

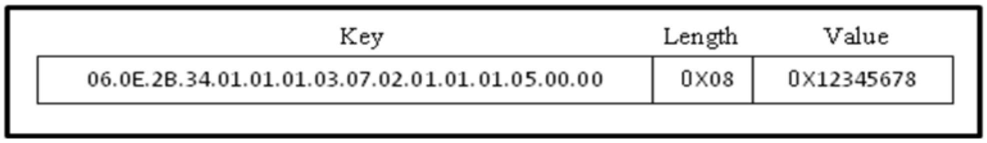
Semi-structured data lies midway between these two. This approach provides structure to the data, but that structure is self-describing. For example, XML is a semi-structured encoding approach. Every element in an XML document has an opening and closing tag. The structure of the document and included elements are defined in an external XML schema document. If the parser can process the contents of the element then it does so. If the parser does not know how to process that element or the element is corrupted, then the processor can just skip that element and continue processing. This combination of structure and flexibility makes semi-structured approaches ideal for large-scale distributed processing systems.

All XML data are encoded using text characters. In addition, each element has both an opening and closing text tag. As a result, XML documents tend to be very large, given the amount of data that they carry. This overhead is unacceptable for real-time and communications-challenged environments. Key-Length-Value (KLV), on the other hand, is designed for these environments. (KLV) is another approach to semi-structured data. KLV data is encoded using efficient binary encoding techniques developed for the Abstract Syntax One (ASN-1) specification. Rather than an opening and closing tag, KLV uses an opening tag followed by a length field and then the data.   KLV management and encoding techniques allow further optimization by reducing or eliminating the key and length fields. The result is a much higher data versus overhead ratio than is typical of XML.

2 Introduction to KLV

The fundamental KLV unit is the **data element**. A KLV data element consists of three parts, the ***key***, the ***length***, and the ***value*** (data). In its most fundamental form, the key is a Universal Label (UL), which is a unique 16-byte value that defines the element. The UL plays a critical role in defining both the structure of the element and the meaning of the data. For example, the UL 06.0E.2B.34.01.01.01.03.07.02.01.01.01.05.00.00 indicates to the parser that this data element is the UNIX Time Stamp element as defined in the MISB registry.

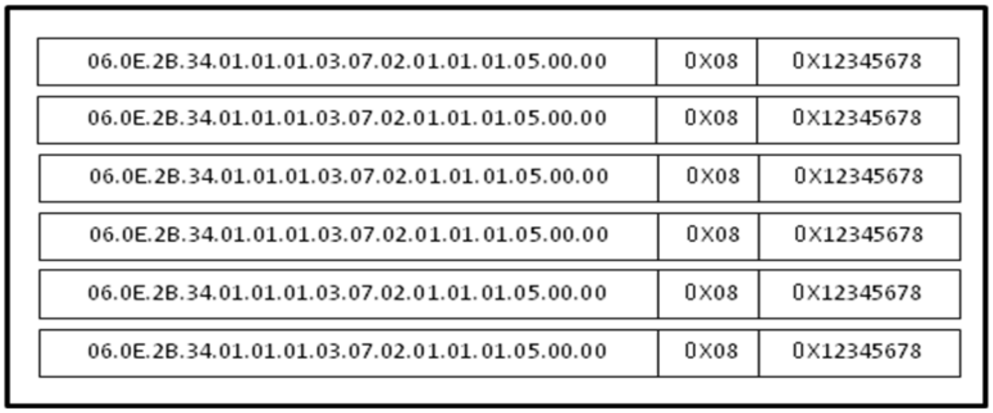
While all KLV elements must map to a UL, the key itself does not have to be a UL (a discussion of alternate key forms is in §5.) Following the key is the length component. The ***length*** value specifies the length in bytes of the ***value*** section. The length value is usually OID encoded (see §6.2) to minimize its size. However, alternate encoding techniques are also allowed (see §6.2). The final component is the ***value*** component.   Value is a sequence of "***length***" bytes of data that are interpreted based on the UL value. In the example of the UNIX Time Stamp (Figure 1), the value is a 64-bit integer that represents microseconds since January 1, 1970.



Key	Length	Value
06.0E.2B.34.01.01.01.03.07.02.01.01.01.05.00.00	0X08	0X12345678

Figure 1: KLV Element Example

KLV data elements are packaged in **KLV Protocol Data Units (PDU)**. Each PDU organizes KLV data elements into a complete body of information. PDUs are usually tagged with a time stamp asserting that this is the data as it existed at a specific date and time. Time stamps also allow correlation of KLV PDUs with other data, such as, video and audio.



06.0E.2B.34.01.01.01.03.07.02.01.01.01.05.00.00	0X08	0X12345678
06.0E.2B.34.01.01.01.03.07.02.01.01.01.05.00.00	0X08	0X12345678
06.0E.2B.34.01.01.01.03.07.02.01.01.01.05.00.00	0X08	0X12345678
06.0E.2B.34.01.01.01.03.07.02.01.01.01.05.00.00	0X08	0X12345678
06.0E.2B.34.01.01.01.03.07.02.01.01.01.05.00.00	0X08	0X12345678
06.0E.2B.34.01.01.01.03.07.02.01.01.01.05.00.00	0X08	0X12345678

Figure 2: KLV PDU Example

A collection of KLV elements and PDUs compose a KLV stream that typically is packaged with motion imagery in a MPEG-2 Transport Stream container and transported over a connectionless protocol such as UDP/IP or RTP/UDP/IP, or directly over RTP/UDP/IP.

Multiplexing techniques, such as MPEG-2 Transport Stream, combine multiple data streams, such as KLV, motion imagery, and audio for carriage as a single entity. Such methods assume a KLV time stamp to correlate KLV metadata with associated video and audio frames. The constituent streams are then de-multiplexed at the receiving end, decoded, and synchronized to provide an integrated picture (voice, video, and metadata) to the viewer.

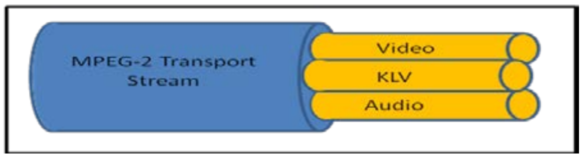


Figure 3: MPEG-2 Transport Stream

3 The Rules

KLV is a self-describing, highly-efficient encoding for streaming data. The following rules assure that any KLV stream is self-describing, highly efficient, and supports streaming transport. Three golden rules are:

Rule #1: **KLV is left-to-right deterministic**: KLV supports streaming transport protocols. As such, it is impossible for the parser to look ahead or back in the data stream. The knowledge necessary to process the next byte must be available to the parser now. Therefore, the KLV stream must always provide the information necessary to process a data element prior to the element itself.

Rule # 2: **KLV uses the minimum overhead necessary:** KLV is designed for real-time and bandwidth challenged applications. It can only fill this role if it is very efficient in its use of bandwidth, memory, and processing power. Well-designed KLV should take every opportunity available to reduce the overhead associated with encoding and processing the data.

Rule #3: **KLV is unambiguous:** Registries external to the data stream maintain the KLV types and keys. These registries provide the authoritative definition of the syntax and semantics of a KLV UL and the associated data element. A KLV parser should never be in doubt as to what a KLV data element means and how to process it. However, that does leave the door open for one additional source of ambiguity. A KLV PDU contains a complete body of information. If a KLV key appears twice in a PDU, which value is the right one? To address this issue the concept of KLV *Context* is needed.

KLV elements are independent instances of information. There is nothing in KLV that states that two KLV elements are associated. However, the structure of the PDU and stream implies associations amongst KLV elements. For example, a Sensor Latitude element and a Sensor Longitude element are not required to be two components describing the sensor position. If both elements appear in the same PDU, however, then that association is implied by their proximity in the data structure. Therefore, if KLV elements have an implied association due to their position in the data structure, then they share the same KLV context. Likewise, a KLV context is a structural element of the KLV data that implies association of its constituent KLV elements. KLV contexts are typically defined by the boundaries of the PDU or, in the case of complex data, the value field of their parent KLV element.

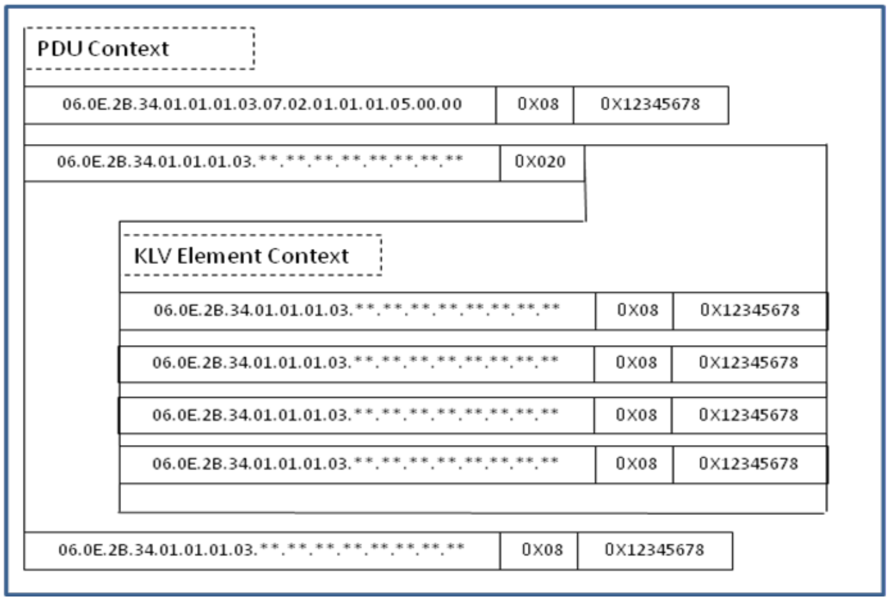

PDU Context
06.0E.2B.34.01.01.01.03.07.02.01.01.01.05.00.00
0X08
0X12345678
06.0E.2B.34.01.01.01.03.**.**.**.**.**.**.**.**
0X020
KLV Element Context
06.0E.2B.34.01.01.01.03.**.**.**.**.**.**.**.**
0X08
0X12345678

Figure 4: KLV Context Example

Having defined the concept of KLV context, a corollary to Rule #3 follows:

Corollary One (1) to Rule #3: a KLV key will appear once and only once in a KLV context.

4 KLV Parsing

KLV is a self-describing encoding method. That does not mean that the KLV stream must carry all of the information needed to process the stream. Rather, it means that there must be sufficient information for the processor to identify the appropriate external information sources and know how to use those sources.

KLV is left-to-right deterministic (Rule #1). The encoding of that information, therefore, must be left to right as well. The KLV key is critical to providing this information. Each KLV key maps to a sixteen (16) byte Universal Label (UL). These labels identify the external resource that manages the supporting information for that element, how the length field is encoded, and (if this is not a simple triplet) the structure of the following KLV data.

For KLV, those external resources are KLV registries. Two registries matter: 1) the Society of Motion Picture and Television Engineers (SMPTE) maintains a registry in its document SMPTE RP 210.10-2007; and 2) the Motion Imagery Standards Board (MISB) maintains a registry on the MISB Web site (www.gwg.nga.mil/misb). Unfortunately, these registries are not machine-readable. Their information content must be built into the encoding and decoding logic.

5 KLV Structures

Rule #2 states that "KLV uses the minimum overhead necessary." Yet, with a sixteen (16) byte key, a one (1) byte length field, and a four (4) byte value, KLV requires twenty-one (21) bytes to encode a four byte (integer) value. That is hardly low overhead. There are, however, a number of established techniques to reduce the overhead of a KLV PDU. In addition, as long as the rules are complied with, other constructs can be built.

KLV structures can reside in the value field of a KLV triplet. There must be at least one full "root" KLV triplet in any PDU. The value field of this triplet can itself be a structure. The length of this "root" element is the length of the enclosed structure. The elements of that structure may in turn contain KLV structures ad infinitum. When fully exploited, a valid KLV PDU might contain only one full sixteen-byte key.

5.1 Universal Data Sets

Universal Data Sets are composed of elements that use a full sixteen-byte UL as the key. This appears to violate Rule #2 – to use the minimum overhead necessary – but there are valid uses for this structure in light of Rule #3 – to be unambiguous. Consider an element for sensor name. Suppose there is more than one sensor within a PDU; under Rule #3 this element can only be used once to avoid ambiguity. However, by packaging the elements for each sensor in its own Universal Data Set the association of sensor name with sensor is unambiguous. In short, the potentially ambiguous element is within its own KLV context, and therefore it is legal under Rule #3.

Advantages:
Groups related data elements, eliminating ambiguity
UL can be constructed from available data with no need to reference outside resources
Disadvantages:
No reduction in overhead

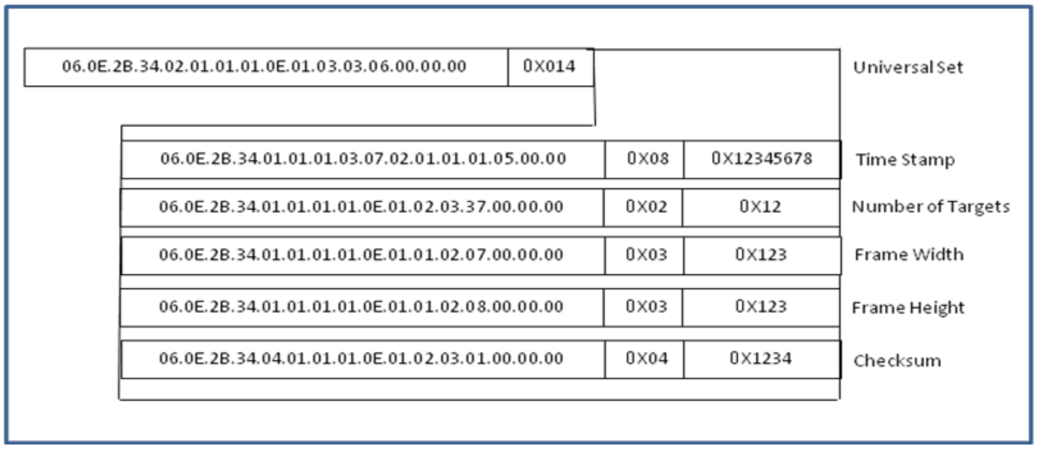

06.0E.2B.34.02.01.01.01.0E.01.03.03.06.00.00.00
0X014
Universal Set
06.0E.2B.34.01.01.01.03.07.02.01.01.01.05.00.00
0X08
0X12345678
Time Stamp
06.0E.2B.34.01.01.01.01.0E.01.02.03.37.00.00.00
0X02
0X12
Number of Targets
06.0E.2B.34.01.01.01.01.0E.01.01.02.07.00.00.00
0X03
0X123
Frame Width
06.0E.2B.34.01.01.01.01.0E.01.01.02.08.00.00.00
0X03
0X123
Frame Height
06.0E.2B.34.04.01.01.01.0E.01.02.03.01.00.00.00
0X04
0X1234
Checksum

Figure 5: Universal Set Example

5.2 Global Data Sets

Global Data Sets reduce the size of KLV keys by taking advantage of the case where all of the ULs of the data elements in a group have the same high-order key values. The resulting keys are the unique subsets of the full sixteen-byte UL. For example, consider a group made from elements that all have ULs that fit the pattern 06.0E.2B.34.01.01.01.01.*.*.*.*.*.*.*.*. The keys of the Global Data Set have to fill in the values only for the missing elements. This approach allows reconstruction of a full UL from the Global Data Set UL and the Global Data Set element keys.

Advantages:
Groups related data elements, eliminating ambiguity.
UL can be constructed from available data with no need to reference outside resources
Reduces overhead
Disadvantages:
Elements of the set must share the same values in the high-order bytes of their ULs.

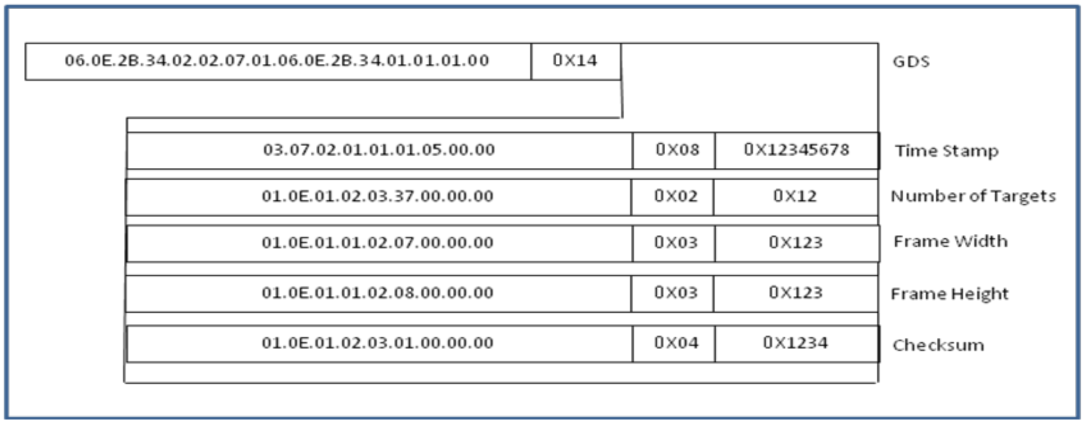

06.0E.2B.34.02.02.07.01.06.0E.2B.34.01.01.01.00
0X14
GDS
03.07.02.01.01.01.05.00.00
0X08
0X12345678
Time Stamp
01.0E.01.02.03.37.00.00.00
0X02
0X12
Number of Targets
01.0E.01.01.02.07.00.00.00
0X03
0X123
Frame Width
01.0E.01.01.02.08.00.00.00
0X03
0X123
Frame Height
01.0E.01.02.03.01.00.00.00
0X04
0X1234
Checksum

Figure 6: Global Data Set Example

5.3 Local Data Sets

Local Data Sets (LDS) go one-step further by moving the UL-to-data-element mapping to an external document. An index value (tag) is assigned to each element in the LDS. Typically, a tag is only one byte long. The controlling document maps each index value to a corresponding UL. To process an LDS element, the parser refers to the controlling document to retrieve the associated UL. It then has the complete UL-Length-Value triplet it needs to proceed with processing.

Advantages:
Groups related data elements, eliminating ambiguity
Reduces overhead through the use of very small tags
Can include any data element in any order
Disadvantages:
The encoder/decoder logic must know or can access at run time the tag-to-UL mappings.

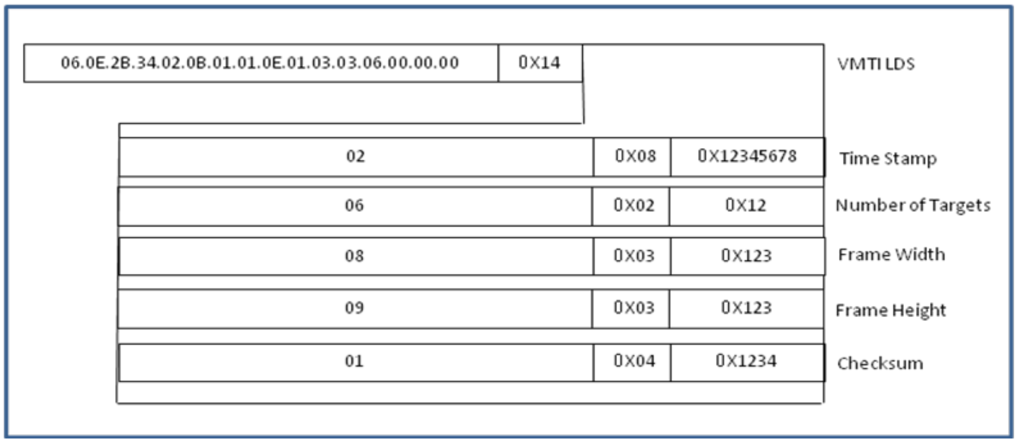


Figure 7: Local Data Set Example

5.4 Variable Length Packs

Variable Length Packs further reduce overhead by eliminating the key altogether. By defining a specific sequence for the elements of the pack[1], the position in that order maps back to the UL for that data element. This approach is similar to that of the LDS except that the tag value is implicit.

Advantages:
Groups related data elements, eliminating ambiguity
Reduces overhead through the elimination of tags
Can include any data element
Disadvantages:
The encoder/decoder logic must know or can access at run time the sequence-to-UL mappings.
All elements must be included and they must be in the specified order, much like “structured”

[1] SMPTE 336m does not formally differentiate between the terms “pack” and “set” (or “data set”). In usage, they are equivalent.

data.

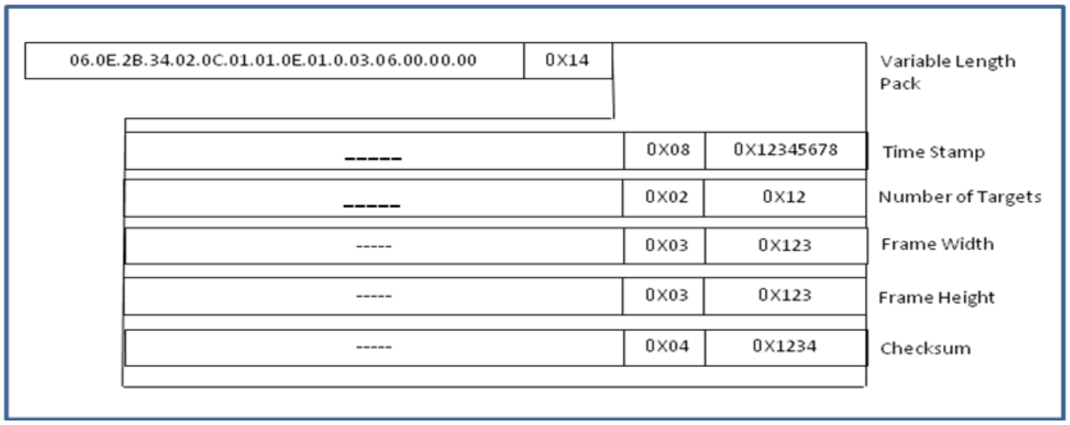


Figure 8: Variable Length Pack Example

5.5 Defined Length Packs

Defined Length Packs build on Variable Length Packs by eliminating the length field. This is accomplished by either defining the length of the element in the controlling document that defines the sequence-to-UL mapping, or by encoding the element in such a way that the length is a part of the encoding process. OID encoding, for example, is a common way to do this.

Advantages:
Groups related data elements, eliminating ambiguity
Reduces overhead through the elimination of tag and length fields
Can include any data element
Disadvantages:
The encoder/decoder logic must know or can access at run time the sequence-to-UL mappings.
The length of each element must be either defined in the controlling document or the element itself must be encoded so as to include the length of that element
All elements must be included, and they must be in the specified order, much like “structured” data.

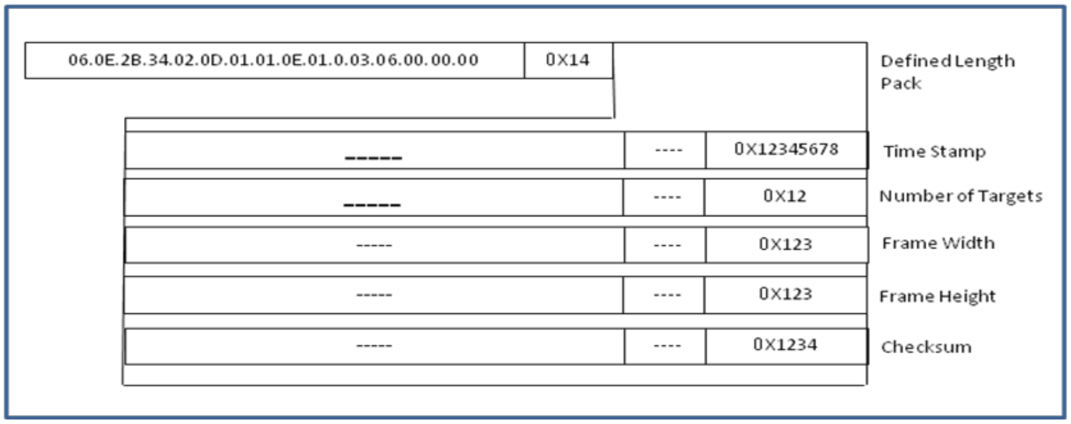


Figure 9: Defined Length Pack Example

5.6 Truncation Packs

Truncation Packs are not formal KLV structures. Rather, they are a technique that takes advantage of KLV encoding to mitigate one of the disadvantages of using Packs. Both Variable Length and Defined Length Packs require that all of the possible elements must be included in each pack. However, there is sufficient information in the encoding of a Pack to remove some of the elements and still comply with the three rules.

Each element of a pack has a length encoded with it or defined in an external document. The sum of those lengths is the total length of the Pack. This value is captured in the length field of the Pack itself. There is no rule that requires that the sum of the expected element lengths and the Pack length field agree. In fact, if we drop the last few elements of a Pack, and update the length to account for just those that are present, the Pack will process just fine. The application of this principal is what is referred to as a Truncation Pack.

The rules for implementing a Truncation Pack are as follows:

1) Start with the elements to be used in a Variable Length or Defined Length Pack.
2) Assign each element of that Pack a priority, from highest to lowest.
3) Populate the elements of the Pack in decreasing priority, optionally omitting one or more of the lowest priority elements. Note that no element may be omitted unless all elements of lower priority are also omitted; "skipping" elements is not permitted.
4) Calculate the total length of the elements that were populated.
5) Store the calculated length in the Pack length field.

Advantages:
Groups related data elements, eliminating ambiguity
Reduces overhead through the elimination of tag and (in defined length packs) length fields
Reduces overhead of populating data elements that would contain no valid information
Can include any data element
Disadvantages:
The encoder/decoder logic must know or can access at run time the sequence-to-UL mappings.
The length of each element must be either defined in the controlling document or the element itself must be encoded so as to include the length of that element
Elements must be defined in order of decreasing priority. If a low priority element is populated, higher priority elements must be populated even if there is no valid information to populate them with.

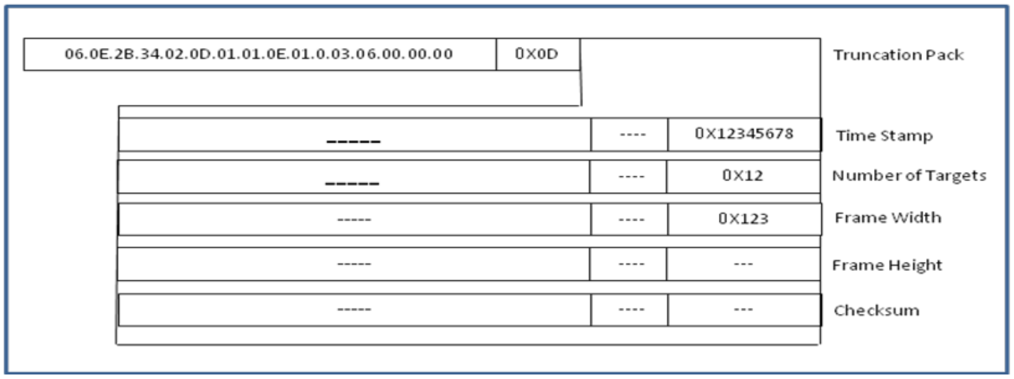


Figure 10: Truncation Pack Example

5.7 Types

To accommodate handling of commonly occurring aggregations of data elements, SMPTE Draft CD2003 provides for the definition of KLV "types". A type is a KLV structure made up of a defined set of related KLV elements. For example, the elements Latitude, Longitude, and Height, which almost always appear together, could be grouped into a type. Type definitions are maintained in a KLV Types Dictionary.

Types are organized into classes. A summary of the most relevant classes are in Table 1.

Class	**Description**
Basic Types	Classic computer science data types ("byte strings")
Enumerated Types	Lists ("enumerations") of specific valid values, similar to C *enum*
Record Types	Fixed structures("records"), similar to C *struct* and XML *sequence*
Multiples	Arrays
Reference Types	"Pointers" to other types
Choice Types	Alternate types, similar C *union* and XML *choice*
Group Types	Collections, similar to C++ *class* and XML *complexType* constructs

Table 1: Classes of KLV Types

A full treatment of KLV types is beyond the scope of this paper, partially because the applicable specifications are still in work.

6 KLV Encoding Techniques

6.1 Universal Labels (UL)

KLV identifies each data element with a sixteen (16) byte Universal Label (UL). The Society of Motion Picture and Television Engineers (SMPTE) or a designated registry authority registers each UL to assure its uniqueness and validity.

Byte	**Field**	**Description**
		Universal Label Header
1	OID	Object identifier – always 0x06
2	UL-Size	16-byte size of the UL – always 0x0E
		UL Designator
3	UL Code	Concatenated sub-identifiers ISO, ORG – always 0x2B
4	SMPTE Designator	SMPTE sub-identifier – always 0x34
5	Category Designator	Category designator identifying the category of registry described
6	Registry Designator	Registry designator identifying the specific registry in a category
7	Structure Designator	Designator of the structure variant within the given registry designator
8	Version Number	Version of the register that first defines the item. The item will appear in that and all later versions.
		Item Designator
9-16	Item Designator	Unique identifier for the particular item within the context of the UL Designator.

Table 2: Universal Label Format

The sixteen-byte UL serves as more than a unique identifier. It also encodes information useful for the processing of the KLV element. A breakdown of that information and how it is encoded in the UL format follows:

Bytes 1 – 4: Bytes one (1) through four (4) are always the same. They designate this UL as being part of the key space managed by the SMPTE.

Byte 5, Category Designator: Byte five (5) identifies the category of the element:

1 = Dictionaries – Definitions of individual data elements.

2 = Groups – Definitions of sets (groups) of data elements, such as a Set or a Pack.

3 = Wrappers and Containers – Definitions of frameworks for collections of information.

4 = Labels – Definitions of descriptions that augment ULs.

Byte 6, Registry Designator: Byte six (6) is divided into bit fields each with its own content.

Bits 0 – 2 (0x00 through 0x07) indicate the type of dictionary, group or container. Table 3 illustrates how bytes five (5) and six (6) work together to identify a resource type.

Byte 5	Byte 6	Byte 7	Meaning
Dictionaries			
0x01	0x01	0x01	Metadata dictionary - SMPTE
0x01	0x01	0x04	Metadata dictionary – MISB
0x01	0x04	0x01	Types dictionary - SMPTE Draft CD2003
Groups (Sets and Packs)			
0x02	0x01		Universal Set – SMPTE 395M
0x02	0x02		Global Set – SMPTE 395M
0x02	0x03		Local Data Set – SMPTE 395M
0x02	0x04		Variable Length Pack – SMPTE 395M
0x02	0x05		Defined Length Pack – SMPTE 395M
Wrappers and Containers			
0x03	0x01		Simple Wrappers and Containers
0x03	0x02		Complex Wrappers and Containers
Labels			
0x04		0x01	Labels Register – SMPTE 400M

Table 3: Universal Label Byte Meanings

Bits 3 - 6 (0x08 through 0x78) indicate how the length field is encoded. The valid values for these bits are shown in Table 4. Since the MISB prefers to use BER-OID encoding for length values, this field will usually be zero (0).

Byte 6, Bits 3 - 6	Encoding method
0x00	BER-OID Encoded (unlimited)
0x01	1 Byte (255 max)
0x02	2 Bytes (65535 max)
0x03	4 Bytes (2^{32} – 1 max)

Table 4: Length Encoding Methods

Byte 7, Structure Designator: Byte seven (7) is specific to the registry. We will not elaborate on this value here.

Byte 8, Version Number: Byte eight (8) is the version number. Direction from SMPTE is that this byte should be ignored.

Bytes 9 and 10, Item Designator: The observant reader may have noticed that the UL for Sets, Packs and Wrappers does not provide any way to determine where the element is registered. Bytes nine (9) and ten (10) have been appropriated for that purpose. If byte 9 is not equal to 0x0E, the UL is registered with SMPTE. If byte nine is 0x0E and byte 10 is 0x01, 0x02 or 0x03, then the UL can be found in the MISB registry. If byte 9 is 0x0E and byte 10 is not 0x01, 0x02 or 0x03, then the element is privately registered somewhere else.

6.2 Basic Encoding Rules (BER)

Rule #2 states that "KLV is left-to-right deterministic." This rule poses a dilemma for the parser that has just received a byte of data. "How many bytes should I read before trying to process this data item?" In many cases, such as the value field of a KLV element, that information has been established ahead of time. That is not always the case. BER encoding is a family of techniques to embed the length of a data item within the item itself. In this way, the parser has sufficient information from the data itself to process the element properly.

BER-OID (BER "Object IDentifier"[2]) encoding allows values of any length to be encoded together with information about the length of the value. The value is encoded using a series of binary octets (bytes) where the 1st bit (msb) of each byte indicates whether or not that byte is the last byte in the series. If the 1st bit is set (1) then the remaining 7 bits form part of the value but there are more bytes to follow. If the 1st bit is not set (0) then the next 7 bits are the least significant 7 bits of the value and this is the last byte in the series.

BER Short Form is used in the length field for data elements shorter than 128 bytes. These fields are represented using a single byte (8 bits). The high-order bit of this element is always set to zero (0). The remaining seven (7) bits contain the length of the following data element. An observant reader will notice that BER Short Form is identical to BER-OID for values less than 128.

BER Long Form is used in the length field for data elements longer than 127 bytes. These fields are represented using a number of bytes with the first byte containing the count of bytes to follow. The high order bit of the first byte is always set to one (1). All of the bits in the following bytes are used.

6.3 Universal Identifier Encoding

The UI is BER-OID encoded. Since the UI is made up of single byte values and each byte has a value less than 0x7F, the high order bit is never used, so the BER-OID encoding is the same as if it weren't encoded at all.

[2] Use of the term "Object IDentifier" (OID) is somewhat unfortunate, because BER-OID encoding can be used for any type of value, not just an OID.

6.4 Bit and Byte Ordering

All KLV data is represented using big-endian encoding – Most Significant Byte (MSB) first. Bytes are big-endian bit encoded – most significant bit (msb) first.

6.5 Floating Point Encoding

KLV does not support the direct encoding of floating-point numbers. Floating-point values must be converted to integer prior to encoding them. SMPTE has no standard for how to perform this conversion. However, scaling floating-point values into suitable integer equivalents is a common practice. This approach takes advantage of advance knowledge of the range of valid values for the number to create an unsigned integer that maintains the desired level of precision while minimizing bandwidth.

A number of MISB documents define the floating point to integer mappings to be used for parameters defined within. Alas, there are some subtle differences among these approaches. To provide consistency, the MISB has developed *RP 1201 Floating Point to Integer Mapping*, which defines a standard approach. The description of float-to-integer mapping that follows should be considered merely illustrative. Consult RP 1201 for an authoritative specification.

Scaled Encoding of Floating Point Values – NOTIONAL:

Scaled encoding takes advantage of three items of information known about the floating-point value:

1) Maximum floating-point value (max_float), the limit beyond which we will not see any valid data.
2) Minimum floating-point value (min_float), the limit below which we will not see any valid data.
3) Encoding integer size (int_size), the number of bits used to encode the floating-point value. [Keep in mind that BER-OID encoding reserves the high order bit of each byte. Thus, for BER-OID encoding, int_size should be a multiple of seven (7), or the bits available from each byte.]

With these values, calculate the encoding scale factor (encode_fctr), decoding scale factor (decode_fctr) and the precision factor (precision_fctr).

```
float_range = max_float – min_float
integer_range = (2 ** int_size) - 1
encode_fctr = integer_range / float_range
decode_fctr = float_range / integer_range
precision_fct = 0.5 * decode_fctr
```

Using these factors, encode the floating-point value f_val as an integer value i_val:

```
i_val = ((f_val – min_float) + precision_fctr) * encode_fctr
```

The first step is to subtract the minimum floating-point value from f_val. This shifts the range of floating-point values to align with the range of unsigned integer values by shifting min-float to align with zero (0).

Next the precision factor is added. When converting a floating-point value to integer, the least significant digits are lost. This has the effect of always rounding down. For example, the floating-point value 7.9 becomes the integer 7 when we drop the fraction. Adding the precision factor causes the conversion to round properly. For example:

```
    7.9+0.5 (the precision factor) = 8.4 which converts to 8 (integer)
    7.4+0.5 = 7 which converts to 7 (integer)
```

Finally, multiply the adjusted floating-point value by the encoding factor to scale the floating-point range to match the unsigned integer range. The scaled and aligned floating-point value is now convertible to unsigned integer.

To convert the integer back into floating-point again:

```
     f_val = (i_val * decode_fctr) + min_float
```

This reverses the previous process by first scaling (i_val * decode_fctr) then shifting (+ min_float) the integer value back into a floating-point value. The precision factor is not applied in this case, since there is no rounding effect in the conversion.